Complexity of Combinator Reduction Machine(Mathematical Foundations of Computer Science and Their Applications)

AUTHOR(S):

Hirokawa, Sachio

数理解析研究所講究録
556巻 1985年 122-146

Complexity of Combinator Reduction Machine

静岡大学工学部情報工学科 広川佐千男(Sachio Hirokawa)

Contents

0. Abstract

The complexity of the computation of recursive programs by the combinator reduction machine is studied. The number of the reduction steps is compared between the two models of computation. The main theorem says that the time required by the reduction machine is linear to that of program scheme. The constant of the linearity was shown to have n order, where n is the maximal arity of functions being used. For analysis of the combinator codes, the notion of extended combinators is introduced.

1. Introduction

D.A. Turner [10,11] showed a new implementation technique for functional programs in terms of combinators. By that method, the

program is compiled to a combinator code and the computation is carried out by graph rewritting called combinator reduction.

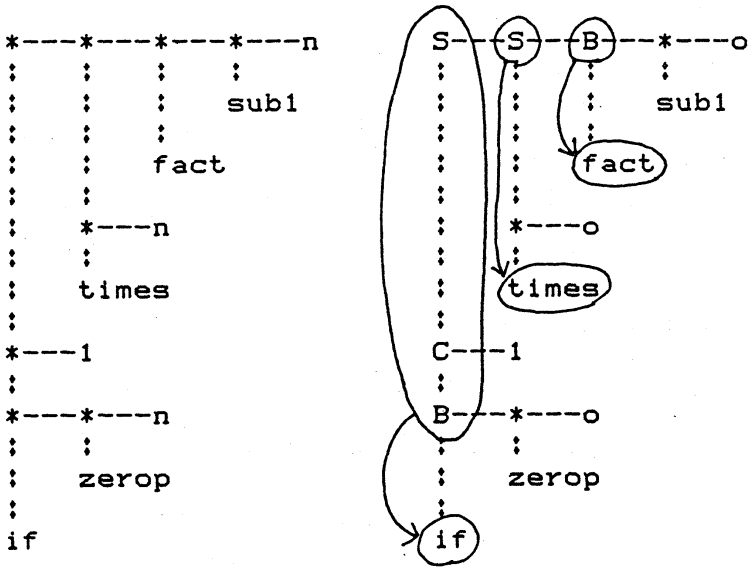
The complexity of the compiled codes and the compiling argorithm has been studied [4, 5, 9,]. They consider rather abstract terms instead of actural programs. Some statistics are obtained in [10] for actual programs concerning to time and space compared with the usual implementation method. But their analysis is only for some examples. This paper is an attempt to give a theoretical foundation for the estimation of the lower/upper bound of the efficiency of combinator reduction machine.

The important merit to use combinators is on the simplicity to realize the call-by-need (lazy evaluation) mechanism. It is known that this evaluation method is optimal [1, 3, 8, 12]. We measure the complexity of the reduction machine, by comparing the number of its reductions and that of the program schemes with optimal reduction. The result does not depent on the structure which the program treats nor on the structure of the program.

The main theorem states that the time required to compute a program by the combinator reduction machine is linear to that of call-by-need computation of program scheme. And the constant of the linearity is shown to be of order n where n is the maximal arity of functions being used in the program.

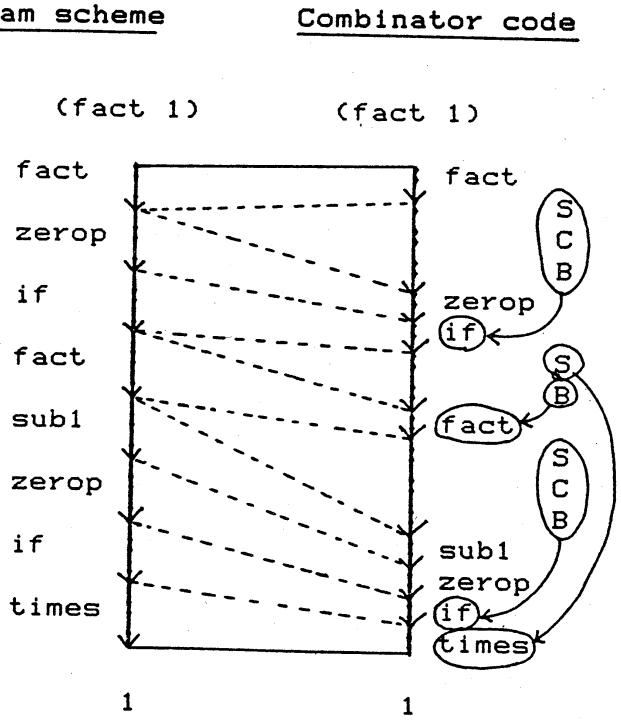
We describe the out line of the analysis by the following example: fact(n) = if(zerop(n),1,times(n,fact(sub1(n)))).
The function body is represented in tree form in the figure below. On the right side of it is the compiled "combinator code"

of the function. Here, the symbols "S","B" and "C" are called combinators. Each combinator is a kind of direction post which shows the path of parameter passing. "S" is for both way. "C" is for the left. And "B" is for the right. Each combinator is assign to some occurrence of a function symbol (the precise definition is given in def 5.1). Note that the number of combinators assigned to a function is at most 3.



Each function and combinator has its "reduction" rule, i.e., graph rewriting rule. The reduction sequences to compute fact(1) = 1, in program scheme reduction and the combinator reduction are shown below. Every function symbol on the left can be found on the right. Here again, Note that the correspondence between combinators and the function is preserved in the combinator

reduction. Thus the total number reductions of S,C and B can be estimated, roughly, by three times of the function reduction. Therefore we can conclude that the cobinator code reduction is at most four time of the program scheme reduction.



Those who are familiar with the combinatory logic[2] and the program scheme [3 , 4 ,] can skip to the section 5, after taking a view of the def 3.3 (extended combinator code), def 3.5, 3.6 (abstraction algorithm), and the lemmas in section 4.

2. Recursive Program Scheme

def 2.1 (Term)

Let $F=\{f_1, f_2, \ldots, f_m\}$ be a set of function symbols, each

symbol f_i is given its arity(f_i)>0. Let $V=\{x_1, \ldots, x_k\}$ be a set of variable symbols and $A=\{a_1, \ldots, a_n\}$ be a set of constan symbols. The set P(A,F,V) of terms is defined inductively by:

(1) Every constant is a term,

(2) Every variable is a term,

(3) If t_i is a term for i=1,...,k and f is a function symbol with arity k, then $f(t_1, \ldots, t_k)$ is a term.

def 2.2 (Recursive Program Scheme)

Let A be a set of constants. Let $G=\{g_1, \ldots, g_m\}$ and $F=\{f_1, \ldots, f_n\}$ be distinct sets of function symbols, whose element is called a primitive function and a user-defined function, respectively. A recursive program scheme Σ is a system of equations

$$\Sigma: \begin{cases} f_i(x_1, \ldots, x_{k_i}) = t_i, \\ i=1,..,n \end{cases}$$

where t_i is a term in $P(A,F\ G,\{x_1, \ldots, x_{k_i}\})$.

In the sequal of the paper, we assume that the set C of constants contains special symbol "t" (true) and "f" (false) and that the set G of primitive function symbols conatains a ternary function symbol "if".

Example 2.3

A={t,f,0,1,2,3,....}

G={if,zerop,times,sub1}

F={fact}

fact(x) = if(zerop(x),1,times(x,fact(sub1(x))))

def 2.4 (Interpretation)

By an interpretation I, each primitive function symbol f_i is associated with a mapping $f_i^I : D_1^i \times \ldots \times D_{k_i}^i \rightarrow D_o^i$, where $D_1^i, \ldots, D_{k_i}^i$ and D_o^i are sets. Each constant symbol a_j is associated with an element a_j^I and a set D_j^I which contains a_j^I.

We assume that the constants "t" and "f" are interpreted as "true" and "false" respectively, which is in the set Bool={true, false}. The tarnary function symbol "if" is interpreted as follows: $if(x,y,z) = \begin{cases} y & \text{if } x \text{ is "true"}, \\ z & \text{if } x \text{ is "false"}. \end{cases}$

def 2.5 (recursive program)

A recursive program is a tuple (Σ, I) of a recursive program scheme: $\Sigma : \begin{cases} f_i(x_1, \ldots, x_{k_i}) = t_i \\ i = 1, \ldots, n \end{cases}$

and an Interpretation I of $F \cup A$:

$$f_i^I : D_1^i \times \ldots \times D_{k_i}^i \rightarrow D_o^i$$

$$a_j^I \in D_j^I.$$

Since we consider a fixed program (Σ, I), we will omit explicit mention of Σ and I.

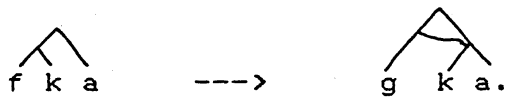
Suppose that we are given an input expression which we want to compute its value with respect to an program. Firstly, the expression is represented by a graph stracture. And the graph is

succesively rewritten according to some rules specified by the program. There are two types of rules, called "expansions" and "simplifications". A simplification is a computation of the form plus(2,3) --> 5. It is a simplification of primitive function call with constants as its arguments. On the othe hand, an expansion is a rewriting of function call of a user-defined function by its body. For example, in the above example, fact(2) is expanded by if(zerop(2),1,times(2,fact(sub1(2)))).

Since we adopt the graph structure to represent the term, the argument is shared and it appears at most once after the expansion. As anothe example, consider a program scheme

$$f(x) = g(x,x).$$

A term f(k(a)) is reduce to g(k(x),k(x)) by expansion of f. In the graph representation it has the form:



Since the argument k(a) of the function f is shared, it has only one occurrence in the resulting graph.

def 2.6 (binary graph)

The data structure on which the computation is performed is the diredted acyclic graph such that

(1) The out degree of each node is 2.

(2) On every leftmost leaf is a function symbol.

(3) Every leave other than (2) has a constant symbol or a variable symbol.

We call such a graph as a binary graph.

def 2.7 (graph represented term)

We represent a term $f(t_1, \ldots, t_n)$ by a binary graph

where t_i^* is the term t_i represented by the same way.

Sometimes, it is written $(..((f\ t_1)\ t_2)..)\ t_n$, or $f\ t_1 t_2 .. t_n$ if the pharenthesis are abbreviated. In such cases, we assume that the pharenthesis are associated from the left.

def 2.8 (substitution)

Let $t, t_1, \ldots, t_n$ be terms and $x_1, \ldots, x_n$ be variables. We denote by $t[x_1/t_1 \ \ldots \ x_n/t_n]$ the term obtained by replacing all the pointers to xi by the pointer to the term t_i for each x_i simaltaneously.

If the term t is the variable x_i itself, then $t[x_1/t_1 .. x_n/ t_n] = t_i$. And when t has no occurence of the variables, we have $t[x_1/t_1 \ldots x_n/t_n] = t$.

def 2.9 (redex, reductum)

Let (Σ, I) be a recursive program:

$$\Sigma: \begin{cases} f_i(x_1, \ldots, x_{k_i}) = t_i, \\ i = 1, \ldots, n. \end{cases}$$

A term is called redex when it is one of the following form.

(1) $f_i(s_1, \ldots, s_{k_i})$, where f_i is a user-defined function,

(2) $g(c_1, \ldots, c_m)$, where g is a primitive function and $c_1, \ldots, c_m$ are constants,

(3) if(v,s_1 , s_2), where v is "t" or "f".

The reductum of each redex is defined according to its type above:

(1) $t[x_1/t_1 \ldots x_{k_i}/t_{k_i}]$

(2) $\mathcal{G}^I(c_1^I,\ldots,c_m^I)$

(3) If v="t" then s_1. If v="f" then s_2.

def 2.10 (P-reduction)

A term t is immediately reduce to t', written t --> t', iff t' is obtained from t by replacing the leftmost outermost redex in t by its reductum. (At the same time, all the pointers to the redex is replaced by the pointer to the reductum.) To spcecify the function symbol of the redex, if it is of the form $f(t_1,\ldots,t_n)$, we write $t \xrightarrow{f} t'$. A one step reduction $t \xrightarrow{f} t'$ is called an expansion of f if f is an unknown function symbol. And it is called a simplification if it is a primitive function symbol.
A reduction is a finite or infinite sequence $d:t_1 \rightarrow t_2 \rightarrow \ldots$.

3. Combinator code and its reduction

In the computation process of a recursive program, a function call of a user-defined function was replaced by its function body in which the acrual parameters are substituted to each variable occurrence. And we assumed that this expansion process is performed by one step.

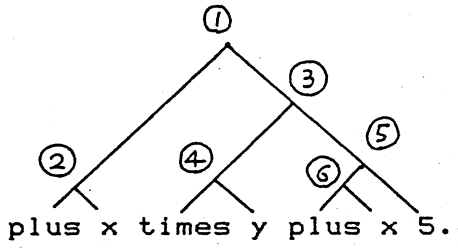
After compiling a program to its "combinator code", the actual arguments of a user-defined function is passed successively to its function body. This process consists of many

steps of "combinator reductions".

A combinator is attached on each node of the function body which is represented by a binary graph. Each combinator shows the way to which possition the parameter has to be delivered. Consider a program

f(x,y) = plus(x,times(y,plus(x,5)))

and its binary graph representation:



When we want to compute the value of f(2,3), the parameter "2" goes over each node as follos:

① : to both subtrees,

② : to the right leaf,

③ : to the right subtree,

④ : to the left subtree,

⑤ : to the right leaf.

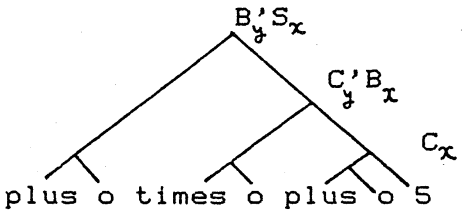
The parameter "3" which corresponds to the variable y is passed as follows:

① : to the right subtree,

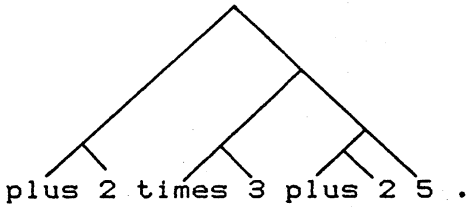
③ : to the left subtree,

④ : to the right leaf.

The combinator code of the function f is



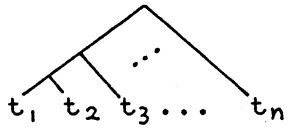
where S,B,C,B' and C' are ther "combinators" which show the way of parameter passing. The combinator S shows that the path branches into two way and that the parameter goes over to the both subtrees. The combinators B and B' tell the way to the right subtree. The combinators C and C' indicate the left branching. By these combinators' direction posts, we can plece the actual parameters to the desired positions and we can reconstruct the term



def 3.1 (combinator code)

A combinator code is a binary graph whose leave has either a constant symbol, a function symbol, a variable or a special symbol, called combinator, I,K,S,B,C,S',B' or C'.
We denote by C(A,F,V) ,or simply C , the set of all combinator codes. Noth that $P \subseteq C$.

To represent a binary graph



in linear form, we write $(..((t_1 t_2)t_3)..t_n)$ or $t_1\ t_2\ t_3 ...t_n$ abbreviating the parenthesis under the assumption that the parenthesis are associated from left.

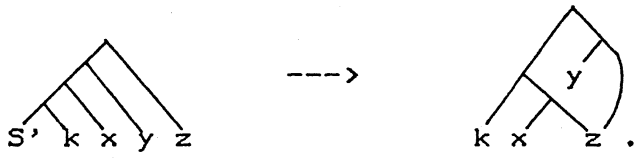
def 3.2 (reduction rules)

Each combinator has the following reduction rules:

Kxy	-->	x
Ix	-->	x
Sxyz	-->	xzyz
Bxyz	-->	x(yz)
Cxyz	-->	xzy
S'kxyz	-->	k(xz)(yz)
B'kxyz	-->	kx(yz)
C'kxyz	-->	k(xz)y

where x,y and z are arbitrary conbinator codes and k is an arbitrary combinator code without variables. The combinator code in the left hand side of each rule is called a redex. The right hand side one is called its reductum.

Each rule shows an graph rewritting rule. The ocurrences of the subtree z are shared on the right hand side of the S and S' rules. Thus the rule for S', for example, is illustrated by



To simplify the analysis of the combinator codes and their reduction process, we introduce a notation to represent the complicated conbinator codes in abbreviated form.

def 3.3 (extended combinator code)

An extended bombinator code is a binary graph such that

(1) Each leaf has either a constant, a function symbol, K, I or a special symbol "o".

(2) The symbol "o" never appears on any left leaf.

(3) Each node is labelled with a finite (possibly empty) sequence of combinators other than K and I.

We write the set of all extended combinator codes as C^* . If we label an empty sequence on each node, any combinator code is regarded as an extended combinator code. In this sense we have $C \subseteq C^*$.

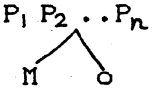
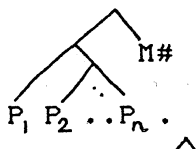
From each combinator code, we can obtain the original combinator code by the following algorithm #. By this transformation we identify the set C* of extended codes and the set C of combinator codes, i.e., C* = C.

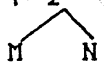
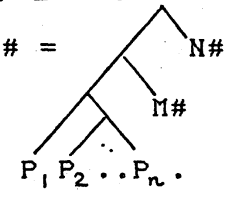
def 3.4 ($C^* \xrightarrow{\#} C$)

Let X be an extended conbinator code. We definde the combinator code X# recursively as follows:

(1) If X is is either a constant, a function symbol or a combinator, then X# = X.

(2) If X =  , then X# = M#.

(3) If X = $P_1 P_2 .. P_n$ and n>0, then X# =  = 

(4) If X = $P_1 P_2 .. P_n$, n>0 and N≠o ,then X# =  = 

def 3.5 (abstraction algorithm)

Ginven a variable x and an extended combinator code t, the abstraction of x from t, denoted by [x]t, is defined by the table below, where t* = [x]t. And it is extended for a sequence of variables $x_1, x_2 \cdots x_n$ and an extented code t by $[x_1 x_2 \cdots x_{n-1} x_n]t = [x_1 x_2 \cdots x_{n-1}]([x_n]t)$.

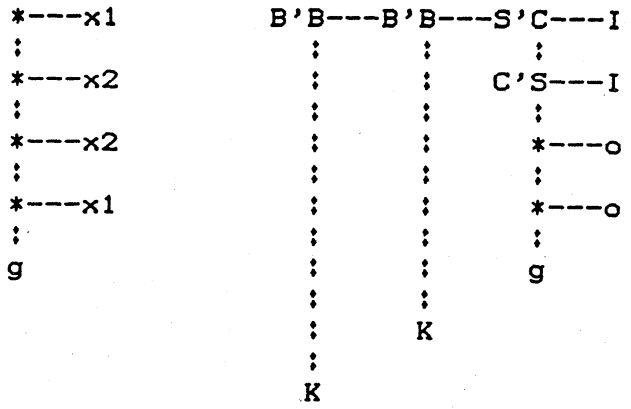
t	t*	condition	Comment
x	I		
M	K M	x: not in M	
M o	M* o	x: in M	
$P_1 \cdots P_n$ M o	$BP_1 \cdots P_n$ M* o	x: in M n >= 0	Extensionality
$P_1 \cdots P_n$ M x	$P_1 \cdots P_n$ M o	x: not in M n >= 0	
M N	S M* N*	x: in M x: in N	
M N	B M N*	x: not in M x: in N	
M N	C M* N	x: in M x: not in N	N ≠ o,x
$P_1 \cdots P_n$ M N	$S'P_1 \cdots P_n$ M* N*	x: in M x: in N	$n \geq 1$
$P_1 \cdots P_n$ M N	$B'P_1 \cdots P_n$ M N*	x: not in M x: in N	
$P_1 \cdots P_n$ M N	$C'P_1 \cdots P_n$ M* N	x: in M x: not in N	

Example 3.6

Consider a term g(x1,x2,x2,x1) and the abstraction of x1, x2, x3 and x4 from it. The combinator code and its extended code are as follows:

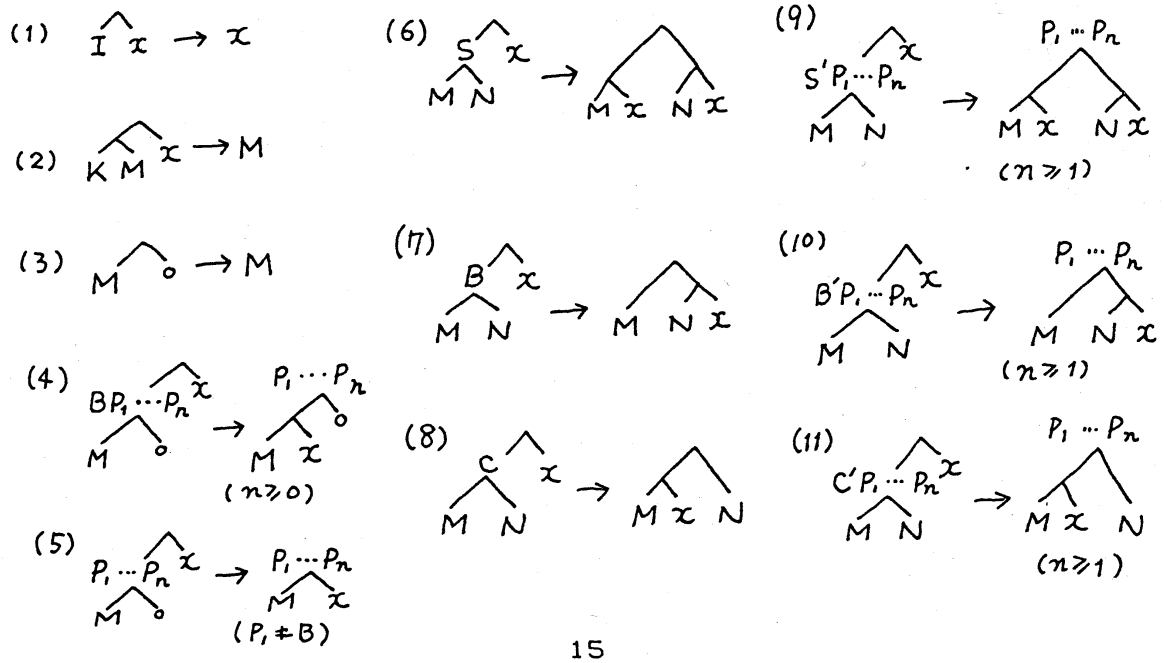
[x1 x2 x3 x4] (g x1 x2 x2 x1)

= (B' B K (B' B K (S' C (C' S g I) I)))



It is possible to use the extended combinator codes as a data structure on which the reduction are performed. Then we should have the corresponding reduction rules as follows:

def 3.7 (reduction rules for C*)

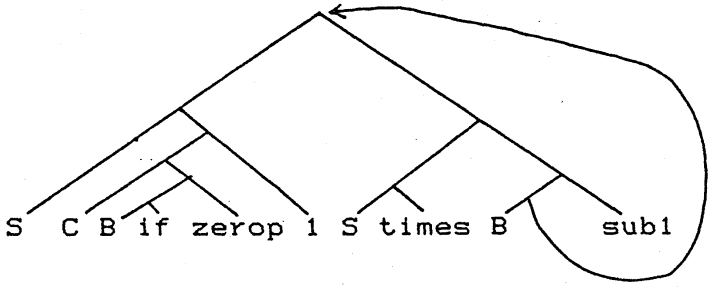


The rules (3) and (4) treats the special symbol "o" and have no corresponding combinator reduction rule. Another fact we can see in the definition of the rules for C* is that the rules for S and S' do the same rewritting. And it is also true for B and B', and C and C'. Therefore the combinators S, B and C are not necessary, if we use the reduction rules for C*. However, we don't go into any more in the present paper.

def 3.8 (C-reduction)

As C-reduction we have the combinator reductions, the simplifications and the expansions. The expansion is different from that of P-reduction. When we expand a user-defined function symbol f followed by its arguments $t_1, \ldots, t_n$, we simply replace the occurrence of the function symbol f by its combinator code $[x_1 \ldots x_n]t$, where t is the body of the function definition. The reduction is carried out to the leftmost outermost redex of the combinator code.

Turner[*10*] uses the cyclic graph for the expansion of a user-defined funciton. For example, the factorial function in example 2.3 has the the cyclic code:



We choose the expansion instead of cyclic graph code for the

simplicity of the analysis. And the similar result holds for the cyclic code representation.

4. Tracing P-reduction from C-reduction

In [10], Turner mentioned the difficulty of tracing C-reduction steps because the code have many combinators. However, if we can eliminate all the combinators from the code, the meaning will become clear. In this section we prove that it is possible and that if we oberve only the of function reductions (i.e., expansions and simplifications) out of the entire C-reduction it is identical to the P-reduction. Thus the P-reduction can be traced from the C-reduction.

The idea of elimination of combinators is as follows. Consider an expansion of a user-defined funciton. In the P-reduction, the parameters are deliverd to the leaves by one step. On the other hand, the function symbol is replaced by a binary graph in C-reduction. On each node of the graph there is a sequence of combinators indicating the path to deliver the arguments. Thus the parameters are passed to the subtrees, not to the leaves. To arrive at every required leaf, all the combinator reductions has to be done. Some of them are not leftmost and is not carried out at this step, because only the left most reductios are permitred in C-reduction. So reducing all the combinator redexes, we leave the C-reduction and reach some stage of the P-reduction.

def 4.1 (t@)

Let t be a combinator code. If there is a combintor code which is reducible from t by combinator reductions and has no combinator redex in itself, then we denote it by t@.

It is not always true that such a code exists. Consider a term Ω= SII(SII), whose combinator reduciton never termninates. When there is such a code, it is unique and it does not depend on the order of reductions, by the Church-Rosser property of Combinatory Logic.

Lemma 4.2 (Correctness of abstraction) Let $M, N_1, \ldots, N_n$ be terms in $\mathbb{P}$ and $x_1, \ldots, x_n$ be variables. Then we have

$((([x_1 \ldots x_n]M)N_1 \ldots N_n)@ = M[x_1/N_1@ \ldots x_n/N_n@]$.

Proof By induction n and the structure of M.

Lemma 4.3 Let t1,t2 be combinator codes and t1 --> t2 in $\mathbb{C}$. If t1@ is defined and t1@ is a term of $\mathbb{P}$, then

(1) t1@ is defined and is a term of $\mathbb{P}$.

(2) If t1 --> t2 by a combinator reduction in $\mathbb{C}$, then t1@ = t2@.

(3) If t1 $\xrightarrow{f}$ t2 by a functiton reduction in $\mathbb{C}$,then t1@ $\xrightarrow{f}$ t2@ in $\mathbb{P}$. Furthermore, if t1 $\xrightarrow{f}$ t2 is leftmost outermost, then t1@ $\xrightarrow{f}$ t2@ is also leftmost outermost.

Proof By induction on the structure of $t \in \mathbb{C}$.

Case 1 $t = fN_1N_2 \ldots N_n$, where f is function symbol and n=arity(f).

Sub-case 1.1 t1 --> t2 is derived from N_i--> N_i'. The lemma is true by induction hypothesis.

Sub-case 1.2 f is a user defined function and t1 $\xrightarrow{f}$ t2 is an expansion of f by its combinator code. Let $f(x_1,\ldots,x_n)=M$ be the definition of the function. Then t2 = $([x_1 \ldots x_n]M)N_1 \ldots N_n$. By Lemma 4.3, we have t2@ = $M[x_1/N_1@ \ \ldots \ x_n/N_n@]$. Therefore t1@ = $f(N_1@)\ldots(N_n@)$ $\xrightarrow{f}$ t2@.

Sub-case 1.3 f is a primitive function. Then N_i is a constant. Therefore we have t1=t1@ ,t2=t2@ and that t1 --> t2 is a $\mathbb{P}$-reduction. So the lemma is true for this case.

Sub-case 1.4 f is "if" and t1=if$N_0N_1N_2$, where $N_0$is either "true" or "false".

Sub-case 1.4.1 N_0="true". Then t2=N . Therefore t1@=if "true" $N_1@N_2@$ --> $N_1@$ = t2@.

Sub-case 1.4.2 N_0="false". Then we have t1@ = if "false" $N_1@$ $N_2@$ --> $N_2@$ = t2@, since t2 = N_2 .

Case 2 t = $PN_1N_2\ldots N_n$,where P is a combinator.

Sub-case 2.1 t1 --> t2 is derived from N_i--> N_i' Then the lemma is true by induction hypothesis.

Sub-case 2.2 t1 --> t2 is the combinator reduction of P. Then we have t1@ = t2@ by the definition of @.

def 4.4 (function reduction part of $\mathbb{C}$-reduction)

Let d : $t_0 \xrightarrow{r_1} t_1 \xrightarrow{r_2} \ldots$ be a $\mathbb{C}$- or $\mathbb{P}$-reduction. We denote be |d| the sequence $r_1r_2\ldots$ of function symbols or combinators. When d is a $\mathbb{C}$-reduciton, the subsequence $r_{i_1}r_{i_2}\ldots$ consiting of all the funciton symbols in |d| is called the function reduction part

of d.

Theorem 4.5 For any term in $\mathbb{P}$, the function reduction part of its $\mathbb{C}$-reduction is identical to its $\mathbb{P}$-reduction.

Proof Since Lemma 4.5 holds, we can prove the theorem by induction on the length of the C-reduction.

5. Structure of C-reduction

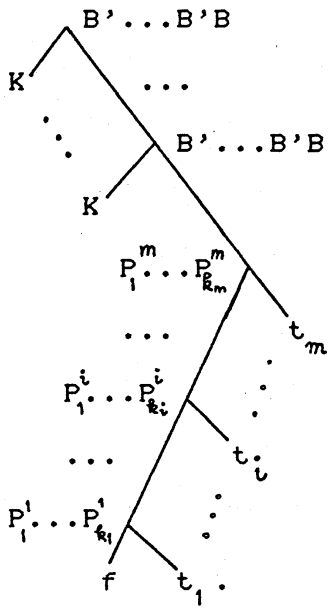
The theorem in the previous section states that the length of the $\mathbb{P}$-reduction is equal to the number of function reductions in the $\mathbb{C}$-reduction. So we can estimate the ratio $\mathbb{C}/\mathbb{P}$ of the length of $\mathbb{C}$-reduction to that of $\mathbb{P}$-reduction by $\mathbb{C}/\mathbb{P} = 1 + Ccom/Cfun$, where Ccom is the number of combinator reductions and Cfun is the number of function reductions in the $\mathbb{C}$-reduction. Thus the problem is boiled down to an analysis of the relation between the function reductions and the combinator reductions in a $\mathbb{C}$-reduction.

To this end, we associate each combinator reduction with a function reduction. And estimate the number of combinator reductions associated to a function reduction.

Through a combinator reduction, an argument soaks into a combinator code which is the compiled code of a user-defined function. After some leftmost combinator reductions, there appears a function symbol at the leftmost position of the code.

If the computation terminates by a constant value, the function symbol disappeara at some stage by its simplification or expansion. We associate the combinator reduction with this function reduction. Since the combinator and the function symbol occurs in the same function body, we only have to analyze the structure of the compiled code. And we prove that the number of occurrences of combinators associated to an occurrence of a function symbol ,in a compiled code, is bounded by some constant which is determined from the arity of the functions being used.

def 5.1 Let f be a function symbol and $M=f(N_1,N_2,..,N_m)$ be a term in P. Let t be the abstruction of $x_1,...,x_n$ from M. Then t has the following form:



Let P be an occurrence of a combinator and g be an occurrence of a function symbol. We write "P < g in t" iff one of the following conditions holds:

(1) g = f and

(i) P is on the left of f, i.e., one of the K, B, B' and P^i_j in the above figure,

or (ii) $P = t_i = I$ for some i.

(2) $P, g \in t_i$ and $P < g$ in t_i for some i.

We say that P precedes g when $P < g$ in t.

Lemma 5.2 Let M be term in P and g be an occurrence of a function symbol in $t = [x_1 \dots x_n]M$. Then we have

$$\text{card}\{P \mid P < g \text{ in } t\} < \frac{1}{4}n^2 + n + mn + m$$

where m is the arity of g.

Proof We count, in the figure of def 5.1, the number of combinator K, B'...B'B, $P^i_1 \dots P^i_{k_i}$, and I separetely.
The combinator K appears after a abstraction of a variable which does not occure in M. Let n_1 be the number of such variables and $n_2 = n - n_1$. Then the number of K's is equal to n_1. At each node, by an abstraction of a variable, the sequence of combinators increases at most by one. Therefore the length of B'...B'B is n_2. Thus the total number of B' and B is $n_1 n_2$. The length of $P^i_1 \dots P^i_{k_i}$ is estimated by n_2, and the number of all P^i_j's amounts to mn_2. And another possiblity is in the case that $t_i = I$. These occurrences of the combinator I is at most m times. Thus we can estimate total number k of combinators such that $P < g$ in t by
$k \leq n_1 + n_1 n_2 + mn_2 + m$. Since $n_1 + n_2 = n$ and $n_i < n$, we have
$k < n + \frac{1}{4}n + mn + m$.

Lemma 5.3 Let x_i and $x_1, \dots, x_n$ be variables and $t = [x_1 \dots x_n]x_i$.

Then card{ P | P is a combinator in t} < $\frac{1}{4}n^2 + n + 1$.

Proof Similar to the proof of Lemma 5.2. Count the number of K, B'..B'B and I in $[x_1 \ldots x_n]x_i$.

Lemma 5.4 Let g be an occurrence of a function symbol in a combinator code of a user-defined function f. Let P be an occurrence of a combinator which precedes g. Let d : $t_0 \dashrightarrow t_1 \dashrightarrow \cdots \dashrightarrow t_n$ there exists the reduction of P in d, then d has the reduction of g.

— be a C-reduction of a term t_0 in P to a constant t_n. If

Proof Since the initial term has no combinators, P comes out after an expansion of f. The function symbol g appears at the same stage in the combinator code of the function. The result of the reduction d is a constant, so the function symbol g has to disappear at some stage. Since g is preceded by P, after the reduction of combinators (execp for the I's) which precedes g, g appears at left most position of the code. Therefore there is the reduction of g in d.

Theorem 5.5 Let t be a term in P. Let com(t) and fun(t) be the number of reduction steps of combinators and functions in the C-reduction of the term. Then we have

$$com(t)/fun(t) \lesssim \frac{5}{4}n^2 + 2n$$

where n is the maximal arity of the functions being used in the program.

Proof Take an arbitrary combinator reduction from the C-reduction of the term. Since t has no combinators at the beginnig, the combinator is introduced by some expansion of a user-defined function f. We associate the combinator reduction with a funciton reduction according to the form of the function body of f.

Case 1. The function body has no function symbol. Then the combinator is associated with the expansion of f. The number of such combinators is not greater than $\frac{1}{4}n^2 + n + 1$ by Lemma 5.4.

Case 2. The funciton body has a function symbol. Then let g be the function symbol proceded by the combinator. (See figure below, where we suppose case that g is the leftmost outermost function symbol and it is a primitive function.) By Lemma 5.3, the C-reduction of t contains the reduction of g. The number of combinators which precedes g is at most $\frac{5}{4}n^2 + 2n$ by Lemma 4.2.

From analysis of both cases, the total number of combinator reductions is less than $(\frac{5}{4}n^2 + 2n)$ times of the number of function reductions.

References

[1] Aiello, L. & Prini, G., An efficient interpreter for the lambda calculus, JCSS 23 (1981) 383-424

[2] Barendregt, H.P., The Lambda calculus, its Syntax and Semantics, North-Holland Pub. Co., 1981

[3] Berry, G. & Levy, J.-J. Minimal and Optimal Computations of Recursive Programs, JACM 26 (1979), 148-175

[4] Burton, F.W., A linear space translation of funcitonal programs to Turner combinators, Inform. Process. Lett., 14 (1982), 201-204

[5] Hikita, T., On average size of Turner's translation to bombinator programs, J. Inform. Process., 7 (1984),

[6] Hughes, R.J.M., Super-combinators, Conf. Rec. of the 1982 ACM Symp. on LISP and Functional Programming, 1982, 1-10

[7] Ida, T., & Konagaya, A., Comparison of closure reduction and combinatory reduction schemes, 1984, preprint

[8] Levy, J.-J. Optimal Reducitons in the lambda calculus, in Seldin & Hindley (eds.), To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980, 159-191

[9] Noshita, K. & Hikita T., The BC-chain method for representing combinators in linear space

[10] Turner, D.A., A new implementation technique for applicative languages, Softw. Pract. Exper., 9 (1979), 31-49

[11] Turner, D.A., Another algorithm for bracjet abstraction, J. Symbolic Logic, 44 (1979), 267-270

[12] Vuillemin, J., Correct and Optimal Implementations of Recursion in a Simple Programming Language, JCSS 9 (1974), 332-354